1996

Software merge: combining changes to decompositions

Berzins, Valdis

Kluwer Academic Publishers

Journal of Systems Integration, 6, 135-150 (1996)
http://hdl.handle.net/10945/51503

MASTER COPY KEEP THIS COPY FOR REPRODUCTION PURPOSES

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
		Reprint

4. TITLE AND SUBTITLE

Software Merge: Combining Changes to Decompositions

5. FUNDING NUMBERS

ARO MIPR 156-94

6. AUTHOR(S)

V. Berzins, D. Dampier

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Computer Science Department
U.S. Naval Postgraduate School
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

U. S. Army Research Office
P. O. Box 12211
Research Triangle Park, NC 27709-2211

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

ARO 30989.13-MA

11. SUPPLEMENTARY NOTES

The view, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Computer aid for software evolution is needed for more effective software development, particularly in contexts where changes to large systems must be made rapidly.

This paper addresses computer aid for the evolution of requirements models and high level software designs. We present an improved method for automatically merging changes to software designs expressed via annotated dataflow diagrams and hierarchical decomposition. This improvement addresses the structure of the design as well as the system behavior the design implies. We also present an improved method for automatically reporting and repairing conflicts between structural changes. These methods can be applied to the informal dataflow diagrams commonly used in requirements modeling and software design as well as to the more specific executable design representations used in the computer-aided prototyping system CAPS.

14. SUBJECT TERMS			15. NUMBER OF PAGES
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Journal of Systems Integration, 6, 135–150 (1996)

Software Merge: Combining Changes to Decompositions

VALDIS BERZINS
Computer Science Department, Naval Postgraduate School, Monterey, California 93943

DAVID A. DAMPIER
Army Research Laboratory, 115 O'Keefe Building, Atlanta, Georgia 30332-0800

ISSN 0925-4676

Journal of Systems Integration

An International Journal

The *Journal of Systems Integration* is a peer-reviewed publication containing original, survey, application, and research papers on all topics related to systems integration. The intent is to encompass a collection of papers that have heretofore been dispersed throughout a wide body of literature involving the interaction of disciplines, technologies, methods and machines necessary to integrate various constituent systems.

The scope of this journal generally parallels the definition of the integration of computer systems. However, it also deals with the general integration of processes and systems, and the development of mechanisms and tools enabling solutions to multidisciplinary problems found in the computer services and manufacturing industries. This journal focuses on the following critical steps found in effective systems integration:

- Process characterization, to understand current process capabilities, behaviors and interfaces,
- Re-engineering and simplification of processes from a system perspective,
- Convergence on a common system architecture with a unified language for data management, and
- Automation of the processes and systems.

Since the succesful implementation of these steps for systems integration requires diverse knowledge bases and expertise in a variety of areas, the journal also emphasizes additional topics such as:

- managing knowledge and information that are physically distributed in various databases,
- computer communications impact on the system process,
- dealing with heterogeneous computers and environments, and
- coordinating diverse computer communiciation networks with information networks.

The aim of the journal is to provide an international and interdisciplinary forum for the dissemination of new theoretical and applied research results, application information and the developments concerning management of systems integration. For instance, it disseminates research work which deals with the problems, issues and solutions of integrated system design, implementation and performance; and with integration technologies that apply to multidisciplinary areas such as computer-aided software engineering, collaborative and distributed systems, and computer integrated manufacturing systems.

Journal of Systems Integration is indexed/abstracted in *INSPEC; Engineering Index; Compendex plus; Ei Page One*

Kluwer Academic Publishers, 101 Philip Drive, Assinippi Park, Norwell, MA 02061, U.S.A.

Journal of Systems Integration, 6, 135–150 (1996)

Software Merge: Combining Changes to Decompositions

VALDIS BERZINS
Computer Science Department, Naval Postgraduate School, Monterey, California 93943

DAVID A. DAMPIER
Army Research Laboratory, 115 O'Keefe Building, Atlanta, Georgia 30332-0800

Received October 27, 1995

Abstract. Computer aid for software evolution is needed for more effective software development, particularly in contexts where changes to large systems must be made rapidly. This paper addresses computer aid for the evolution of requirements models and high level software designs. We present an improved method for automatically merging changes to software designs expressed via annotated dataflow diagrams and hierarchical decomposition. This improvement addresses the structure of the design as well as the system behavior the design implies. We also present an improved method for automatically reporting and repairing conflicts between structural changes. These methods can be applied to the informal dataflow diagrams commonly used in requirements modeling and software design as well as to the more specific executable design representations used in the computer-aided prototyping system CAPS.

Keywords:

1. Introduction

Our goal is to provide computer aid for software evolution, particularly in the critical early stages of determining requirements and software architectures. Expected benefits include automated assistance for combining different changes to a proposed design, assessing their consistency and reconciling structural conflicts between changes. Reliable tool support for change-merging would also facilitate distributing design tasks on a large software development project to a group of engineers working concurrently.

The technology to achieve this is emerging [6, 8], and a significant amount of work has been done on developing change-merging models and building automated change-merging tools for imperative languages [3, 7, 17, 26], as well as for data-flow languages [2, 9, 10, 11, 12] and specification languages [4, 5].

This paper presents an improved change merging method for hierarchies of annotated dataflow diagrams. This method addresses design structure in addition to system behavior. The results apply to a range of notations that have been widely used in requirements analysis, design of software architectures, and computer-aided software prototyping. Our results have a potential impact on tool support for evolution in all of these contexts.

Software evolution dominates many aspects of software development, and tools that support aspects of evolution such as change merging are useful in a variety of situations. Change merging is needed to coordinate concurrent effects of teams of designers, particularly in the contexts of developing large systems and rapid prototyping. Changes to several different

aspects of a large system are often developed concurrently, because it would take too long develop the changes one after another. During iterative development of software prototypes, alternative versions of a prototype are often developed, each of which contains a portion of the desired capability. Because these prototypes can be large, tools that automatically determine the differences between the alternative versions and produce a new version exhibiting the significant behavior modifications from each are desirable.

Software change-merging is also applicable to software maintenance activities [13]. If a software system has been developed using the computer-aided prototyping system (CAPS) or other languages and systems that provide a semantically safe change-merging capability, different versions of that software can be automatically updated with changes made to the base version via a change-merging tool.

Other potential uses of this technology are in the areas of software reuse and reengineering. In software reuse, complex reusable components that contain more functionality than is required for the application can be retrieved from the software repository. The desired functionality can be isolated using graph slicing by taking the slice of the complex component with respect to the output streams desired. The resultant slice will contain all parts of the complex component that affects those output streams.

In reengineering, if a program written in some high-level language can be translated into the prototyping language PSDL, then the evolution support capabilities of CAPS [1, 20, 21] can be applied to update designs in which significant subsystems are realized by legacy code. In addition to enabling parallel exploration of different enhancements to a legacy system that exists in a single configuration, a change-merging capability could be useful in propagating enhancements from a base version of a software family to all of the other configurations in the family. For example, each configuration in the family could be tailored to a different operating environment.

In [11], a change-merging method was defined that is semantics-based and guarantees that if a conflict-free result is produced, it is semantically correct. This is significant because it allows different pieces of a large prototype to be developed independently and integrated with complete assurance of the correctness of the integration. An initial implementation of this method is described in [12]. This tool uses dataflow graph slicing, analogous to program slicing [25], to determine the affected parts of an enhanced version that are different from the base version. It then combines the affected parts of the modified versions with the preserved part of the base version to produce the change-merged version.

A drawback of the initial method is that it works only on flat graphs with no hierarchical decomposition. This was done to guarantee correct slicing. In practice, software prototypes are designed using hierarchical decomposition to provide better understanding and maintainability. The current implementation of the method can expand such designs into equivalent flat graphs, but currently provides no automated way for aggregating the change-merged graphs back into a hierarchical structure consistent with the designer's original intent. The result is executable, but not understandable, because the original design structure is lost. This prevents the tool from being used on very large software projects, the very projects it was intended to support.

An automated method for reconstructing the hierarchical design structure is needed to make change-merging more attractive. Such a method is the main contribution of this

paper. We provide a model and algorithm for automatic reconstruction of decomposition hierarchies for change-merged graphs. This extension to the previously developed method increases automation support for the prototype designer, and makes the results more understandable.

The method was originally developed to support rapid prototyping, because the issues of repeated changes and multiple versions are particularly pressing in that context. The method is also applicable in other contexts, such as supporting the evolution of requirements models and architectures for large software systems.

2. Motivating Context: Computer-Aided Prototyping

Computer-aided prototyping is an evolutionary software development paradigm that overcomes the weaknesses of traditional software development methods by increasing customer interaction during the requirements engineering phase of development. It provides executable specifications that can be evaluated for conformance to real-time requirements, and produces a production software system in a fraction of the time required using traditional methods. Rapid prototyping allows the user to get a better understanding of requirements early in the conceptual design phase of development. It relies on software tools to rapidly create and demonstrate concrete executable models of selected aspects of a proposed system to enable users to assess and validate the model early in the development process. The prototype is rapidly reworked and redemonstrated to the user over several iterations until the designer and users have confidence in a precise view of what the system should do. This process produces a validated set of requirements which become the basis for designing the final product [20].

The prototype can also be transformed into part of the final product. In prototyping methods like the one supported by CAPS, the prototype is an executable shell of the final system, containing a subset of the system's ultimate functionality and a high level representation of the software architecture for the final product. After the design of the prototype is approved by the customer, the missing functionality is added and the system is delivered. In this approach to rapid prototyping, software systems can be delivered incrementally as parts of the system become fully operational.

Change-merging is an integral part of the rapid prototyping method. In the prototype design part of the process, multiple variations of a large prototype are likely to be developed. This can happen when different development teams are working on enhancing different aspects of a system, or when different possible solutions to a problem are explored in different ways. In the first case, it will certainly be necessary for the separately developed pieces of the prototype to be combined into a single system before execution for the customer. In the second case, the customer may desire a system containing some or all of the aspects contained in each solution. In this case, these different prototypes must be sliced to capture the significant parts of each variation and then change-merged to materialize the desired combination. Our change-merging method will allow these combinations to be done automatically, ensuring that the resultant prototype is semantically consistent with all of the input variations. If the changes to the semantics of the prototype are not compatible, then our method will identify the parts of the prototype containing the conflicts. This technology

encourages the designer to explore different solutions to a problem, and to spread the development workload in a large project with more confidence in the subsequent integration of these independent efforts.

3. Prototyping System Description Language

The change-merging method described in [12] and extended here has been implemented for use in the CAPS development system. It is designed to operate on software prototypes written in the Prototyping System Description Language (PSDL) associated with CAPS. PSDL is a high level specification and design language which can be translated into executable code.

PSDL represents software systems as generalized dataflow diagrams annotated with timing and control constraints [19]. The notation is executable and has a formal semantics [18] that is a compatible refinement of informal dataflow diagrams traditionally used in software design. A PSDL prototype is a hierarchical network of components. Each component consists of two parts: a specification and an implementation. The specification of a component defines its interface, and the implementation contains either a PSDL graph implementation, or a reference to a programming language implementation. The PSDL graph implementation contains a set of operators, a set of data streams through which the operators communicate with one another, and a set of control and timing constraints which specify restrictions on the execution of the operators or data streams. These graph implementations are constructed using a top-down approach, where each level of decomposition refines a composite operator at the previous level. This decomposition provides understandability for complex designs. The programming language implementation is written in any high-level programming language like Ada or C that is supported by the environment. A more complete description of PSDL is available in [19], and computational models for PSDL are available in [12, 18, 22].

4. Ancestor Chain Model of Design Structure

We would like to apply past approaches to software change-merging to the domain of software decomposition structures. These approaches work on special kinds of lattices that are also Brouwerian or Boolean algebras [7]. To apply these approaches, we need a refinement ordering on the set of all possible designs. Several examples of refinement relations are shown in Figure 1. Possible types of refinements include: (1) adding additional operators to the prototype, (2) decomposing an atomic operator into a composite operator, (3) grouping related atomic operators under a composite operator, and (4) providing a name for a composite operator. A difficulty is that most of the intuitively acceptable refinement orderings on software decomposition structures fail to qualify as lattices because they have multiple minimal upper bounds, as shown in Figure 2. Nodes 3-6 are all minimal upper bounds for nodes 1 and 2. Nodes 3 and 4 are proper and completely defined decomposition structures whose existence and correspondence to reality are not in doubt. Their greatest lower bound is node 7, which fails to be a common upper bound for nodes 1 and 2. This

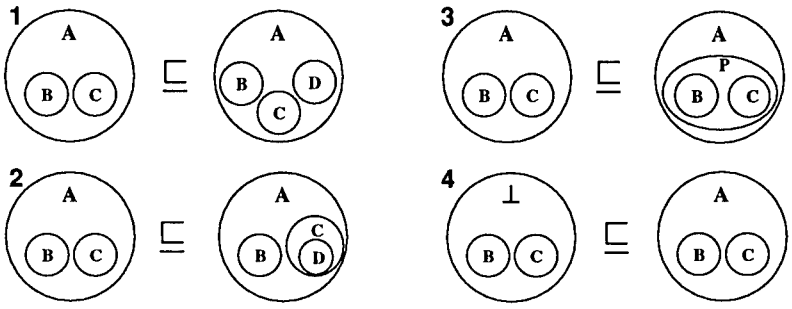


Figure 1. Examples of possible refinements of program designs.

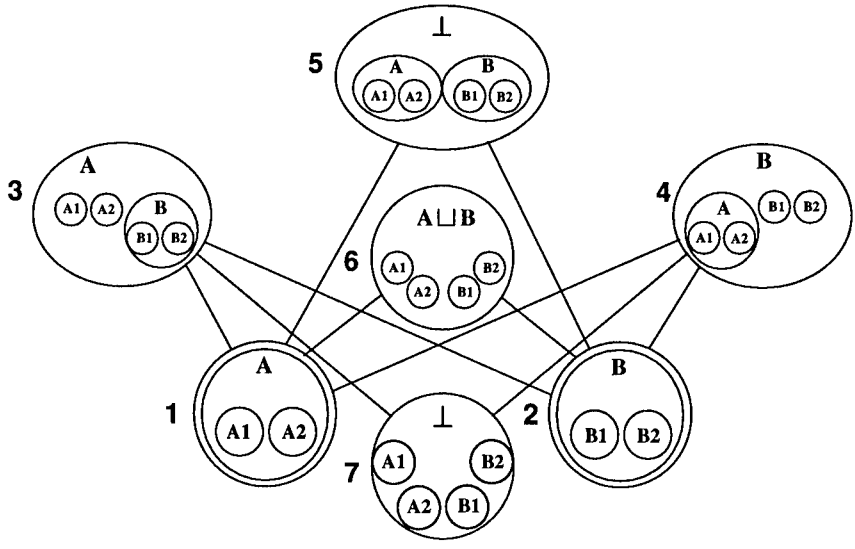


Figure 2. Multiple minimal upper bounds on program designs.

demonstrates that nodes 1 and 2 cannot have a least upper bound. Having no least upper bound means a unique automated choice of the preferred design structure is not possible in the general case.

This implies that the most obvious model of the problem is inadequate and that we need to consider different points of view on the nature of the information contained in a software decomposition. The key insight that leads to a solution is that the essential information about a software decomposition is not an attribute of the substructure of a component, as

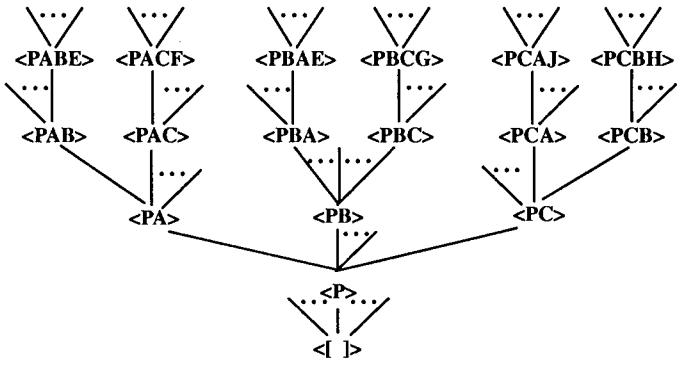


Figure 3. Refinement ordering $\mathcal{R}$ over ancestor chains.

was assumed in developing the refinement ordering illustrated in Figure 3, but rather an attribute of its *context*, namely the location of the component in the hierarchy.

This insight leads to the *ancestor chain* model of software structure which is the basis for our change-merging method. In this model, the position of a component in a design hierarchy is represented as a sequence of ancestor names, where the sequence (A, B, C, D) means that the parent of the current node is D, D's parent is C, C's parent is B, and the top level operator is A. We call these sequences *ancestor chains.*

We define the domain of proper ancestors to be the set of all finite sequences of components, partially ordered by the prefix ordering. Formally this ordering is defined by $x \sqsubseteq y \Leftrightarrow \exists z[y = x \smile z]$, where $\smile$ denotes concatenation of sequences. An example of such a refinement ordering is shown in Figure 3. This ordering has the structure of a tree, with its root at the empty sequence. The tree has infinite depth. Its branching factor at each node is equal to the number of all possible module names; it is conventional to assume that this is infinite as well. This partially ordered set is a *meet semi-lattice* that has greatest lower bounds, but lacks least upper bounds for pairs of sequences where one is not a prefix of the other. This incompleteness in the abstract model corresponds to concrete reality for informal dataflow diagrams as well as for the CAPS system: since PSDL does not allow an operator to have more than one parent, two ancestors in different ancestor chains cannot have a least upper bound that is a legal program, or a proper element of the domain of software decompositions.

We note that the bottom element of the semi-lattice has a double interpretation. The empty sequence serves as a proper element when it is used to represent the ancestor chain of the root component of the decomposition. It also has the more familiar purpose of representing undefined/incomplete information in its role as the ancestor chain of all possible components that are not yet included in a given design decomposition.

To do change-merging over this refinement ordering, we embed it in a lattice like the one shown in Figure 4. We do this by adding improper data elements representing least upper

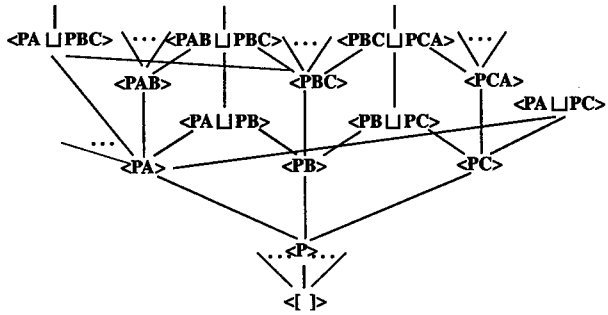


Figure 4. Extended ancestor lattice.

bounds for all sets of incomparable proper elements and adding them to the set. These improper elements (e.g. $\langle a \rangle \sqcup \langle b \rangle$) represent merging conflicts that can be considered to be abstract representations of error messages.

Since our goal is to define a change-merging operation, we extend the domain of proper ancestors in a way that makes it easy to add a pseudo-difference operation that makes the resulting lattice into a Brouwerian algebra. The standard way to do this is to work with downwards-closed sets in the partial ordering (i.e., sets S for which $x \in S \,\&\, a \sqsubseteq x \Rightarrow a \in S$), where the ordering $\sqsubseteq$ denotes the sequence prefix ordering illustrated in Figure 3. Figure 4 abbreviates the set representation by showing only the maximal elements of each set with respect to the $\sqsubseteq$ ordering.

We define the domain of extended ancestors as follows. The elements of the extended domain are downwards-closed sets of proper ancestor chains, where each set represents the least upper bound of all the proper elements in the set. A proper element of the extended domain represents a proper ancestor chain, and consists of a set containing that proper ancestor chain and all of its prefixes. The ordering of the extended domain is the subset relation. This ordering produces a full lattice structure. The corresponding least upper bounds are set unions, and the corresponding greatest lower bounds are set intersections. The lattice operations are well defined because the union and intersection of two downwards-closed sets are both downwards closed.

We note that the ordering on extended ancestors agrees exactly with the sequence prefix ordering on proper ancestors, and that the greatest lower bounds and the least upper bounds also agree with those in the sequence prefix ordering in the cases where these bounds exist in the sequence prefix ordering.[1] Thus this construction gives a consistent extension of the sequence prefix ordering, and the proper ancestor domain can be embedded in the extended ancestor domain.

The pseudo-difference operation $\dot{-}$ that comes with this construction is the downwards closure $\mathcal{DC}$ of the set difference. The pseudo-difference and downwards closure operations are defined as follows.

Data and Operations	Proper Ancestor Chains	Improper Ancestor Chains
x	$\langle ab \rangle$	$\langle a \rangle \sqcup \langle c \rangle$
$\mathcal{R}(x)$	$\{\langle ab \rangle, \langle a \rangle, \langle\rangle\}$	$\{\langle a \rangle, \langle c \rangle, \langle\rangle\}$
y	$\langle a \rangle$	$\langle ab \rangle \sqcup \langle d \rangle$
$\mathcal{R}(y)$	$\{\langle a \rangle, \langle\rangle\}$	$\{\langle ab \rangle, \langle a \rangle, \langle d \rangle, \langle\rangle\}$
$\mathcal{R}(x) \cup \mathcal{R}(y) = \mathcal{R}(x \sqcup y)$	$\{\langle ab \rangle, \langle a \rangle, \langle\rangle\} = \mathcal{R}(\langle ab \rangle)$	$\{\langle ab \rangle, \langle a \rangle, \langle c \rangle, \langle d \rangle, \langle\rangle\} = \mathcal{R}(\langle ab \rangle \sqcup \langle c \rangle \sqcup \langle d \rangle)$
$\mathcal{R}(x) \cap \mathcal{R}(y) = \mathcal{R}(x \sqcap y)$	$\{\langle a \rangle, \langle\rangle\} = \mathcal{R}(\langle a \rangle)$	$\{\langle a \rangle, \langle\rangle\} = \mathcal{R}(\langle a \rangle)$
$\mathcal{R}(x) - \mathcal{R}(y)$	$\{\langle ab \rangle\}$	$\{\langle c \rangle\}$
$\mathcal{DC}(\mathcal{R}(x) - \mathcal{R}(y)) = \mathcal{R}(x \dot{-} y)$	$\{\langle ab \rangle, \langle a \rangle, \langle\rangle\} = \mathcal{R}(\langle ab \rangle)$	$\{\langle c \rangle, \langle\rangle\} = \mathcal{R}(\langle c \rangle)$

Figure 5. Examples of lattice operations in the set representation.

$$x \dot{-} y = \mathcal{DC}(x - y)$$
$$\mathcal{DC}(S) = \{a : Ancestor \mid \exists x[x \in S \,\&\, a \sqsubseteq x]\}$$

It is well known that the downwards closed sets ordered by set inclusion form a Brouwerian algebra with respect to a pseudo-difference operation defined in this way ([23], Theorem 1.14). A definition of Browerian algebras is given in the appendix and a discussion of some of the known properties of Brouwerian algebras and pseudo-difference operations can be found in [7, 24]. An executable specification of the lattice and change-merging operations is given in Figures 6 and 7. These specifications are expressed in OBJ3 [14, 15, 16].

The domain *Component_id* contains unique names for components; its definition is not shown because it has no interesting properties. The domain *Component* contains only proper components, with a trivial ordering: all distinct components are incomparable. The domain *Ancestor* contains only proper ancestor chains, which are finite sequences of components ordered by the prefix ordering. These proper domains are extended with artificial elements that are least upper bounds of finite sets of proper elements and represent merging conflicts. The extended domains are denoted by *Component!* and *Ancestor!*. The OBJ3 keyword *prec* declares the relative precedences of the infix operators; lower numbers indicate tighter binding. The equations define the lattice ordering and the meet and join operations for the extended domains.

The change-merging operation $a[b]c$ represents the result of combining the change from the base version b to the enhancement a with the change from b that results in a different enhancement c. Some examples of software decomposition merges implied by these equations are shown in Figure 8. The ancestor chain merge for node d is not shown in the figure because node d is not present in the merged implementation graph, so that its position in the hierarchy does not have to be computed by the decomposition merging algorithm.

obj DECOMPOSITION_LATTICE **is**
protecting COMPONENT_ID .
sorts Component Component! Ancestors Ancestors! .
subsort Component < Component! < Ancestors! .
subsort Component < Ancestors < Ancestors! .
op C : Component_id → Component .
*** Constructor for building components from component_ids
op ⊥ : → Ancestors .
*** An empty ancestor list, for root components and unused components.
op _ , _ : Ancestors Ancestors → Ancestors [assoc id: ⊥ prec 1] .
op _ , _ : Ancestors! Ancestors! → Ancestors! [assoc id: ⊥ prec 1] .
*** Ancestor list.
op _ ⊑ _ : Ancestors! Ancestors! → Bool [prec 4] .
*** Lattice Ordering.
op _ ⊔ _ : Component Component → Component! [comm prec 2] . *** assoc
op _ ⊔ _ : Ancestors! Ancestors! → Ancestors! [comm prec 2] . *** assoc
*** Least Upper Bound.
op _ ⊓ _ : Ancestors! Ancestors! → Ancestors! [assoc comm prec 2] .
*** Greatest Lower Bound.

vars $C\ C'\ C''$: Component . **vars** $EC\ EC'\ EC''$: Component! .
vars $A\ A'\ A''$: Ancestors . **vars** $EA\ EA'\ EA''$: Ancestors! .

*** Ordering
eq $C \sqsubseteq C' = C == C'$.
eq $A \sqsubseteq EA \sqcup EA' = A \sqsubseteq EA$ or $A \sqsubseteq EA'$.
eq $EA \sqcup EA' \sqsubseteq EA'' = EA \sqsubseteq EA'$ and $EA' \sqsubseteq EA''$.
eq $\bot \sqsubseteq EA$ =true .
eq $EA \sqsubseteq \bot = EA == \bot$.
eq $(EC, EA) \sqsubseteq (EC', EA') =$ if $EC \sqsubseteq EC'$ then $EA \sqsubseteq EA'$ else false fi .
*** Least Upper Bounds
eq $((EC \sqcup EC'), EA) = (EC, EA) \sqcup (EC', EA)$.
eq $(EC, (EA \sqcup EA')) = (EC, EA) \sqcup (EC, EA')$.
cq $EA \sqcup EA' = EA'$ if $EA \sqsubseteq EA'$.
*** Greatest Lower Bounds
eq $C \sqcap C' =$ if $C == C'$ then C else $\bot$ fi .
eq $EA \sqcap (EA' \sqcup EA'') = (EA \sqcap EA') \sqcup (EA \sqcap EA'')$.
eq $\bot \sqcap EA = \bot$.
eq $(EC, EA) \sqcap (EC', EA') = ((EC \sqcap EC'), (EA \sqcap EA'))$.

endo

Figure 6. OBJ3 equations for constructing ancestor lattice.

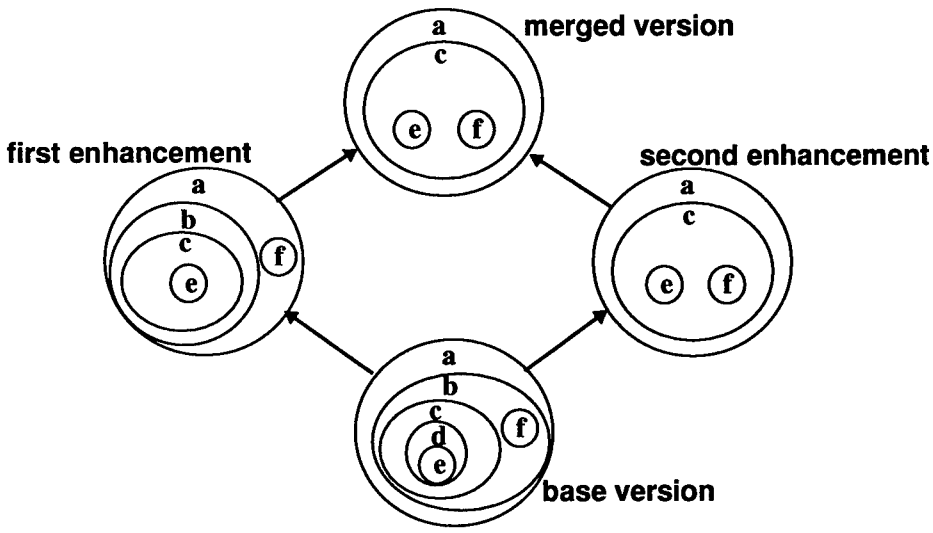
```
obj DECOMPOSITION_CHANGE_MERGING is
   protecting DECOMPOSITION_LATTICE .
   op _ ÷ _ : Ancestors! Ancestors! → Ancestors! [prec 3] . *** Pseudo-difference
   op _ [ _ ] _ : Ancestors! Ancestors! Ancestors! → Ancestors! [prec 4] . *** Merge

   vars C C' C'' : Component .      vars EC EC' EC'' : Component! .
   vars A A' A'' : Ancestors .      vars EA EA' EA'' : Ancestors! .

      *** Pseudo-Difference
   eqA÷EA = if A ⊑ EA then ⊥ else A fi .
   eq(EA ⊔ EA')÷EA'' = (EA÷EA'') ⊔ (EA'÷EA'') .
      *** Merge
   eqEA[EA']EA'' = (EA÷EA') ⊔ (EA ⊓ EA'') ⊔ (EA''÷EA') .

endo
```

Figure 7. OBJ3 equations for merging ancestor chains.

5. Change-Merging Algorithm

The initial algorithm for semantics-based change-merging of flattened PSDL prototypes was described briefly in [11], and exhaustively in [12]. This algorithm takes three PSDL prototypes as input; a base version and two modified versions. The change-merge operation applied to the implementation graphs uses graph slicing to identify the preserved part of the base and the affected parts of the modified versions. To do the graph slicing accurately and guarantee that all dependencies are found, the hierarchically decomposed graphs are expanded, yielding equivalent flat implementation graphs. The result of the change-merge is a PSDL program with a completely expanded flat implementation graph. The initial algorithm produced a correct merged graph whenever no conflicts were reported. This graph was converted into a PSDL program that can be translated into an executable representation of the merged version and is useful for demonstrating the behavior of the merged version. However, the designer's original decomposition was lost in the process, so the result of the merge was not a suitable basis for human review or further enhancement.

This section presents an extension of the initial PSDL change-merging algorithm based on the ancestor chain model of section 4. The extended algorithm does everything the initial algorithm does, then separately determines the design structures of the three versions, combines them, and uses the result to transform the merged flat graph back into a new hierarchical design. The final step of the initial algorithm constructs a PSDL prototype from the merged graph in a subprogram called *build_prototype*. In the extended algorithm, this subprogram is replaced with the subprogram *decompose_graph*, shown in Figure 9. This function operates on the *change_merged* program produced by the original algorithm along



$$
\begin{array}{lll}
a: & \langle\rangle[\langle\rangle]\langle\rangle & = \langle\rangle \\
b: & \langle a\rangle[\langle a\rangle]\langle\rangle & = \langle\rangle \\
c: & \langle ab\rangle[\langle ab\rangle]\langle a\rangle & = \langle a\rangle \\
e: & \langle abc\rangle[\langle abcd\rangle]\langle ac\rangle & = (\langle abc\rangle \div \langle abcd\rangle) \sqcup (\langle abc\rangle \sqcap \langle ac\rangle) \sqcup (\langle ac\rangle \div \langle abcd\rangle) \\
 & & = \bot \sqcup \langle a\rangle \sqcup \langle ac\rangle \\
 & & = \langle ac\rangle \\
f: & \langle a\rangle[\langle ab\rangle]\langle ac\rangle & = (\langle a\rangle \div \langle ab\rangle) \sqcup (\langle a\rangle \sqcap \langle ac\rangle) \sqcup (\langle ac\rangle \div \langle ab\rangle) \\
 & & = \bot \sqcup \langle a\rangle \sqcup \langle ac\rangle \\
 & & = \langle ac\rangle
\end{array}
$$

Figure 8. Examples of change-merging ancestor chains.

with the three given versions of the implementation graphs as input and produces a new *psdl_program* with a hierarchically decomposed graph.

The *decompose_graph* subprogram uses an array indexed by the nodes in the graph to hold the ancestor chain for each node. Each element of the array is initialized by the function *merge_ancestor_chains*, which calculates the merged ancestor chain for each node in the merged flat graph according to the equations in Figure 7. The ancestor chain of each node with respect to each of the three given versions of the PSDL program is determined by the function *find_ancestor_chain* which recursively searches the tree of graphs in each of the given decomposition structures until it finds the node, and then determines the ancestor chain by retracing the path back up to the root of the tree, recording each parent in the chain on the way. When the loop terminates, the array holds the merged ancestor

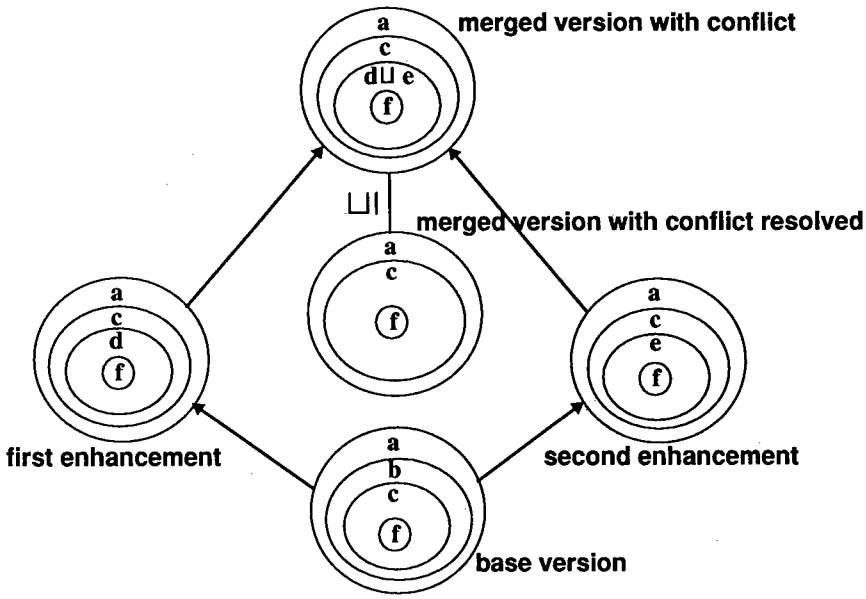
```
Algorithm decompose_graph(MERGE: in psdl_program; A, BASE, B: in psdl_graph)
return psdl_program;
  ANCESTORS: array(node_id) of extended_ancestor;

 MERGE_CHAIN, A_CHAIN, B_CHAIN, BASE_CHAIN: extended_ancestor;
 NEW_PSDL: psdl_program;
  begin

    for every node N in MERGE loop
        A_CHAIN := find_ancestor_chain(A, N);
        B_CHAIN := find_ancestor_chain(B, N);
        BASE_CHAIN := find_ancestor_chain(BASE, N);
        merge_ancestor_chains(A_CHAIN,BASE_CHAIN,B_CHAIN,MERGE_CHAIN);
        ANCESTORS(N) := MERGE_CHAIN;
    end loop;
    report_conflicts(ANCESTORS);
    ANCESTORS := resolve_conflicts(ANCESTORS);
    NEW_PSDL := reconstruct_prototype(MERGE, ANCESTORS);
    return NEW_PSDL;
 end decompose_graph;
```

Figure 9. Algorithm *decompose_graph*.

chains, which can be either proper sequences or improper ancestor chains that are least upper bounds of two incompatible proper ancestor chains. The subprogram for reporting conflicts *report_conflicts* scans the array of ancestor chains and reports and describes a conflict for each of the improper chains in the array, if there are any. The subprogram *resolve_conflicts* repairs any conflicts in the graph according to the methods described in the next section, resulting in an array that contains only proper ancestor chains. The last subprogram *reconstruct_prototype* builds the new graph according to the recreated design structure and replaces the flat graph in the merged prototype with the new graph.

6. Conflict Resolution and Error Messages

Hierarchical decompositions provide grouping information. The grouping information aids human understanding, but does not affect function. This implies that partial recovery of the designer's intent can be acceptable, as long as the recovered information is compatible with all of the decisions that were made. Even in cases where changes to the grouping information implicit in the decomposition structures conflict, sensible results can always be produced by taking the maximal proper grouping that is consistent with the result of the merge in the cases where it produces an improper element of the ancestor chain lattice representing a merging conflict. This element can be computed conveniently from the



$$\begin{aligned}\langle acd\rangle[\langle ac\rangle]\langle ace\rangle &= (\langle acd\rangle \dot{-} \langle ac\rangle) \sqcup (\langle acd\rangle \sqcap \langle ace\rangle) \sqcup (\langle ace\rangle \dot{-} \langle ac\rangle) \\ &= \langle acd\rangle \sqcup \langle ac\rangle \sqcup \langle ace\rangle \\ &= \langle acd\rangle \sqcup \langle ace\rangle \text{ (***conflict***)} = \langle ac(d \sqcup e)\rangle\end{aligned}$$

$$\langle acd\rangle \sqcap \langle ace\rangle = \langle ac\rangle. \text{ (***resolution of conflict***)}$$

Figure 10. An example of a resolved conflict.

normal form for ancestor chains defined by the equations in Figure 6. In this normal form each lattice element is represented as the least upper bound of a set of one or more proper ancestor chains. For practical applications, the size of this set will be at most one more than the number of merging operations that were used to construct the lattice element, and if conflicts are resolved by each merging operation, the set will have at most two elements. If the result of merging ancestor chains is a conflict term, compute the maximal compatible proper ancestor chain by replacing all *least upper bounds* in the normal form with *greatest lower bounds* and simplifying. The result is the strongest proper term consistent with both changes. An example is shown in Figure 10.

The change-merging method for decomposition structures given in this paper is a total operation that always produces a result. Sometimes the result is a proper element, which represents a conflict-free merge, and sometimes it is an improper element, which represents

a conflict between incompatible changes. The equations given in Figure 6 can be used as rewrite rules to transform all improper elements into a normal form that looks like the least upper bound of a set of irredundant proper elements. This representation can be used to produce informative error messages.

The basic structure of our ancestor chain model associates context descriptions with each operator. Consequently, the error message can specify *which operator* has a design structuring conflict. The normal form specifies exactly which decisions about the placement of the operator in the hierarchy conflict, so that this information can be provided with the error message. Finally, in some cases the nature of the conflict can be further localized, by using the following equation to further transform conflict terms:

$$(C,\ EA) \sqcup (C,\ EA') = (C,\ (EA \sqcup EA'))$$

This transformation factors out common prefixes of improper ancestor chains, thus getting rid of information that does not contribute to the conflict and producing a more specific error diagnosis. The results of doing this are shown both algebraically and graphically in Figure 10. The error message resulting from the transformed conflict term for the example can be rendered in English as "structural conflict: both D and E are required to be direct parents of operator F".

7. Conclusion

Our main result is an extension to the change-merging algorithm of [11] that preserves the significant design structure as well as the significant behavior of the given versions. This is an improvement over the previous method because the automatically constructed merge can be used as a basis for further analysis and prototype enhancement as well as for execution and prototype demonstration. The previous algorithm produces a version that is suitable only for execution and demonstration, because the structure of the design is lost by the transformation it applies. This paper shows how that design structure can be recovered after the behavioral transformation is applied.

Our change-merging method and algorithm are formulated in terms of PSDL, the prototyping language used by the CAPS system. Because the PSDL model is a generalization and extension of the informal dataflow diagrams, the same change-merging method can also be applied to the informal dataflow diagrams commonly used in software requirements and software design. Our results can therefore be used to extend the degree of computer support for many variations of this widely used notation, such as those of Yourdon and DeMarco.

A supplementary result of the paper is an approach to error diagnosis and repair that applies to change-merging conflicts. The approach exploits the improper data elements introduced by the extension of the proper data domain to the Brouwerian algebra needed to support the change-merging domain. The improper data elements are used both for providing a specific description of the nature of the conflict, and for deriving the most informative conflict-free design structure that is compatible with the overconstrained value representing the conflict. We note that the Brouwerian algebra introduced in [24] to better

explain the algorithm for merging changes to while-programs given in [17] also contains analogous improper data elements, and conjecture that these elements can also be exploited to support improved conflict diagnosis and possibly some form of conflict resolution.

Appendix—Definitions

This appendix contains formal definitions for some of the standard algebraic concepts used in the body of the paper.

Definition 1. A **lattice** is an algebra $\langle L, \sqcup, \sqcap \rangle$ such that the set L is closed under the operations $\sqcup$ and $\sqcap$ and the following properties are satisfied for all x, y, and z in L:

- $x \sqcap x = x$ and $x \sqcup x = x$
- $x \sqcap y = y \sqcap x$ and $x \sqcup y = y \sqcup x$
- $x \sqcap (y \sqcap z) = (x \sqcap y) \sqcap z$ and $x \sqcup (y \sqcup z) = (x \sqcup y) \sqcup z$
- $x \sqcap (x \sqcup y) = x$ and $x \sqcup (x \sqcap y) = x$

Definition 2. The partial ordering $\sqsubseteq$ associated with a lattice is defined by $x \sqsubseteq y \Leftrightarrow x \sqcap y = x$ for all x, y in L.

Definition 3. A **Browerian algebra** [23] is an algebra $\langle L, \sqcup, \sqcap, \dot{-}, \top \rangle$ such that

- $\langle L, \sqcup, \sqcap \rangle$ is a lattice with the greatest element $\top$,
- The set L is closed under the operation $\dot{-}$, and
- $x \dot{-} y \sqsubseteq z \Leftrightarrow x \sqsubseteq y \sqcup z$ for all x, y, z in L.

Notes

1. The least upper bound $x \sqcup y$ with respect to the sequence prefix ordering exists if and only if $x \sqsubseteq y$ or $y \sqsubseteq x$. In the first case $x \sqcup y = y$ and in the second $x \sqcup y = x$. The greatest lower bound $x \sqcap y$ always exists and is the longest common prefix of the sequences x and y.

References

1. S. Badr, and Luqi, "Automation support for concurrent software engineering," *Proceedings of the 6th International Conference on Software Engineering and Knowledge Engineering*, Jurmala, Latvia, June 20–23, 1994, pp. 46–53.
2. V. Berzins, "On merging software extensions," *Acta Informatica*, Springer-Verlag, pp. 607–619, 1986.
3. V. Berzins, "Software merge: Models and methods for combining changes to programs," *Journal of Systems Integration* 1(2), pp. 121–141, August 1991.
4. V. Berzins, and Luqi, *Software Engineering With Abstractions*, Addison-Wesley, 1991.

5. V. Berzins, Luqi, and A. Yehudai, "Using transformations in specification-based prototyping." *IEEE Transactions on Software Engineering* 19(5), pp. 436–452, May 1993.
6. V. Berzins, *Proceedings of the ARO/AFOSR/ONR Workshop on Increasing the Practical Impact of Formal Methods for Computer-Aided Software Development: Software Slicing, Merging and Integration.* Monterey, California, October 1993.
7. V. Berzins, "Software merge: Semantics of combining changes to programs." *ACM Transactions on Programming Languages and Systems* 16(6), pp. 1875–1903, November 1994.
8. V. Berzins, *IEEE Computer Society Press Tutorial: Software Merging and Slicing.* IEEE Computer Society Press, 1995.
9. D. Dampier, "A model for merging different versions of a PSDL program," Master's Thesis, Naval Postgraduate School, Monterey, California, June 1990.
10. D. Dampier, and Luqi, "A model for merging software prototypes," Naval Postgraduate School Technical Report CS-92-014, 1992.
11. D. Dampier, Luqi, and V. Berzins, "Automated merging of software prototypes." *Journal of Systems Integration* 4(1), Kluwer, January 1994.
12. D. Dampier, "A formal method for semantics-based change-merging of software prototypes," Ph.D. Dissertation, Naval Postgraduate School, Monterey, California, June 1994.
13. D. Dampier, R. Byrnes, and M. Kindl, "Computer-aided maintenance for embedded real-time software," *Proceedings of the 19th Army Science Conference*, Orlando, Florida, June 1994.
14. K. Futatsugi, J. Goguen, J. Jouannaud, and J. Meseguer, "Principles of OBJ2", *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, ACM, New Orleans, 1985, pp. 52–66.
15. J. Goguen, and J. Meseguer, "Rapid prototyping in the OBJ executable specification language." *Software Engineering Notes* 7(5), pp. 75-84, December 1982.
16. J. Goguen, et al., *Introducing OBJ*, SRI Technical Report SRI-CSL-88-8, August 1988.
17. S. Horwitz, J. Prins, and T. Reps, "Integrating non-interfering versions of programs," *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages*, Association for Computing Machinery, New York, New York, January 13–15, 1988.
18. B. Kramer, Luqi, and V. Berzins, "Compositional semantics of a real-time prototyping language." *IEEE Transactions on Software Engineering* 19(5), pp. 453-477, May 1993.
19. Luqi, V. Berzins, and R. Yeh, "A prototyping language for real time software." *IEEE Transactions on Software Engineering*, pp. 1409–1423, October 1988.
20. Luqi, "Software evolution through rapid prototyping." *IEEE Computer*, May 1989.
21. Luqi, "A graph model for software evolution." *IEEE Transaction on Software Engineering* 16(8), August 1990.
22. Luqi, "Real-time constraints in a rapid prototyping language." *Journal of Computer Languages* 18(2), pp. 77–103, Spring 1993.
23. J. McKinsey, and A. Tarski, "On closed elements in closure algebras." *Annals of Mathematics* 47(1), pp. 122–162, January 1946.
24. T. Reps, "On the algebraic properties of program integration." *Science of Computer Programming* 17(1–3), pp. 139–215, December 1991.
25. M. Weiser, "Program slicing." *IEEE Transactions on Software Engineering*, pp. 352–357, July 1984.
26. W. Yang, S. Horwitz, and T. Reps, "A program integration algorithm that accommodates semantics-preserving transformations," *Proceedings of the 4th ACM SIGSOFT Symposium on Software Development Environments*, Irvine, California, December, 1990, pp. 133–143.

For information about current subscription rates and prices for back volumes for
Journal of Systems Integration, ISSN 0925-4676
please contact one of the customer service departments of Kluwer Academic Publishers or return the form overleaf to:

Kluwer Academic Publishers, Customer Service, P.O. Box 322, 3300 AH Dordrecht, the Netherlands, Telephone (+31) 78 524 400, Fax (+31) 78 183 273, Email: services@wkap.nl

or

Kluwer Academic Publishers, Customer Service, P.O. Box 358, Accord Station, Hingham MA 02018-0358, USA, Telephone (1) 617 871 6600, Fax (1) 617 871 6528, Email: kluwer@world.std.com

Call for papers

Authors wishing to submit papers related to any of the themes or topics covered by *Journal of Systems Integration* are cordially invited to prepare their manuscript following the 'Instructions for Authors'. Please request these instructions using the card below.

Author response card

Journal of Systems Integration

I intend to submit an article on the following topic:

_____

_____

Please send me detailed 'Instructions for Authors'.

NAME : _____

INSTITUTE : _____

DEPARTMENT : _____

ADDRESS : _____

_____

_____

Telephone : _____

Telefax : _____

Email : _____

Library Recommendation Form

Route via Interdepartmental Mail

To the Serials Librarian at: _____

From: _____ Dept./Faculty of: _____

Dear Librarian,

I would like to recommend our library to carry a subscription to
Journal of Systems Integration, ISSN 0925-4676
published by Kluwer Academic Publishers.

Signed: _____ Date: _____

Request for information about current subscription rates and prices for back volumes of

Journal of Systems Integration, ISSN 0925-4676

Please fill in and return to:

Kluwer Academic Publishers, Customer Service, P.O. Box 322, 3300 AH Dordrecht, the Netherlands

Kluwer Academic Publishers, Customer Service, P.O. Box 358, Accord Station, Hingham MA 02018-0358, USA

☐ Please send information about current program and prices
☐ Please send a free sample copy

NAME :
INSTITUTE :
DEPARTMENT :
ADDRESS :
Telephone :
Telefax :
Email :

REF. OPC

STAMP

Journal of Systems Integration
Kluwer Academic Publishers,
101 Philip Drive
Assinippi Park
Norwell, MA 02061
U.S.A.

TO : The Library
FROM: _____

VIA INTERDEPARTMENTAL MAIL